Introduction to Verilog-HDL

Tomohisa Uchida (E-sys, IPNS, KEK), 2014/12/01
Translated by Yun-Tsung Lai (E-sys, IPNS, KEK), 2024/06/20

Introduction

This is an introductory book explaining the minimum necessary Verilog-HDL syntax for people without experience in circuit design.
Some books from commercial market is written for experts of circuit design, so they may be difficult for beginners. The purpose of this document is to be able to express circuits in HDL anyway without explaining convenient or elegant descriptions to improve work efficiency. It is recommended to read it first, then learn with commercial books.

Prior knowledge

- What are block diagrams and circuit diagrams
- Hierarchical circuit concept
- Operation of the basic 4-element AND, OR, INV gates and D-FF.
- Synchronous circuit design
- State machine

In this document, we will proceed on the design method for synchronous circuit design and the simulation.

Table of contents

Reference

We referred to the following documents:

- 小林 優、“入門 Verilog-HDL 記述 ”、CQ 出版
- 小林 優、デザインウェーブ付録 “はじめてでも使える HDL 文法ガイド” 、CQ 出版
- 枝 均、 “Verilog-HDL によるテストベンチ”、テクノプレス
- 安岡 貴志、“デジタルデザインテクノロジ 10 月増刊号 Verilog HDL & VHDL テストベンチ記述の初歩” 、CQ 出版

Disclaimer

1. Structure of Verilog-HDL

Overview of Verilog-HDL.

Module

Verilog-HDL describes circuit information with text files. In addition, circuit blocks are expressed in units called "modules". Verilog-HDL is a collection of modules. Modules are distinguished by circuit function names called "module names".

- Modules can be divided freely by designers.
- Modules are often divided into functional units.
- Modules can have a hierarchical structure.
- The module can be freely named by designers.
- A module is written in a text file
- The usual extension of the file is ".v".
- We recommend one circuit block (module) in one file.
- In syntax, it is possible to write multiple circuit blocks in one file, but it is better to write one circuit block in one file for easier understanding.
- Use half-width characters.
- Although full-width characters can be used in comments, some tools may behave strangely, so please be careful when using them.

A module is a file describing a circuit. You can think that the circuit (operation) is described by HDL.

About hierarchical structure

Since modules can have a hierarchical structure, we can do the same way as a schematic diagram with a hierarchical structure.
We can call another module inside one module. The same module can be called (copied, reused) from multiple modules.
It is defined as a circuit only when it is incorporated (excluding the top module at the highest level).
For example, if you use two module-B in module-A, module-B is used in two places inside module-A as shown below.

```
module-A -+- module-B
          |
          +- module-B
```

In this case, the two module-B operate as independent circuits. We need to distinguish between them. Therefore, when calling them, number the circuits as shown below. It is the same as the reference number (part number) of the PCB board.

```
```

```
module-A(TOP) -+- module-B(B1)
               |
               +- module-B(B2)
```

The module number is in parentheses. This module number is called an "instance name" in Verilog-HDL.
To summarize, a module have two names. They are the "module name" and "instance name".

About naming

Verilog-HDL uses various kinds of names, such as module names, instance names, signal names, etc. There are rules for naming.

Identifier: Names are called identifiers in principle. Regular identifiers have the following rules:

- The first character has to be half-width alphabetic or an underscore.
- Identifiers consist of half-width letters, numbers, and underscores.
- Case sensitive.
- Using keywords are not allowed.

Keywords are the words reserved for Verilog-HDL syntax. For example, assign, module, etc.

Numerical value

Digital circuits are binary arithmetic circuits, so they often process numbers. Various radix representations can be used.

```
[bit width]'[base][value]
```

To represent a value, first is the bit width (positive integer), single quotes, radix, and value, in this order.
First is the bit width. Different from software, hardware needs to consider the number of signal lines (bit width) used.
Next, put single quotes.
Radixes are represented using the symbols below.

Symbol	Base
b or B	2
o or O	8
d or D	10
h or H	16

For instance, for a value 128, bit width is 8, hex:

```
8'h80
```

For a value 128, bit width is 8, decimal:

```
8'd128
```

You can ignore some parts, but we recommend to write everything carefully at first.

Special numbers

The following values are provided for simulation.

Symbol	Base
z or Z	High impedance
x or X	Not fixed

High impedance, or called 3-state, the signal state not being driven.

2. Structure of module

The features of Velirog-HDL code:

- Written in a text file
- The extension is usually ".v"
- Contain one or more modules
- Although multiple modules can be written in one text file, one module is often written in one text file.

Structure of module

For a module, it starts from "module" and ends with "endmodule". The code is written in between.

```
module module_name (port list);

  port declaration: module's I/O signals' definition

  parameter declaration:
  Property of internal signals: driven by a register or not

  Circuit descriptions
    Instance of lower module
    assign statement, always statement, etc

end module
```

For instance, a module "MY_XOR2" with 2-input XOR:

```
/* This module is an example of 2-input XOR.
This line is a comment.
*/
module MY_XOR2(
  input IN_A, // input A
  input IN_B, // input B
  output O // output O
);

//-----------------------------------------------------------
// XOR
//-----------------------------------------------------------
wire IA; // Output of a combinational circuit is defined as wire
wire IB;

assign IA = IN_A & ~IN_B;
assign IB = ~IN_A & IN_B;
```

```
assign O = IA | IB;

endmodule
```

This code will be explained in order:

Comment

Two types of expressions for comment:

- From /* to */. It can span multiple lines.
- // to end of a line

Module name and port list

```
module MY_XOR2(
  input IN_A, // input A
  input IN_B, // input B
  output O // outputO
);
```

For a module, it starts with "module", then a space, then the module name "MY_XOR2".
Between the parenthesis, there is the port list, where the input/output definitions and signal names of this module MY_XOR2 are defined.
Let's see the first signal as an example.

```
  input IN_A, // input A
```

Here, "input" represents the I/O definition and IN_A represents the signal name.
In the port list, write the I/O definition on the left, enter a space, and write the signal name on the right. Separate each signal with a comma.

3 types of I/O definition:

Keyword	Meaning
input	input
output	output
inout	inout (bi-directional)

Notice for FPGA design:

Input/output signals cannot be used inside the FPGA. Otherwise, the development tool will automatically convert it into independent input and output signals. Of course, it can be used as an I/F signal for the outside of FPGA.

Designers can freely assign names for signals, but they must follow identifier rules. There are three signals here: IN_A, IN_B, O. Separate each signal with a comma. Anything from // to the end of the line is interpreted as a comment. Comments can also be expressed by using /* and */.
Verilog-HDL interprets units separated by semicolons. Therefore, a semicolon is always needed at the end of a sentence. Everything up to a semicolon as one sentence, so you can insert any tabs and spaces as you'd like. You can write in a way that is easy for you to read.

For example, you can also write as below.

```
module MY_XOR2(input IN_A, input IN_B, output O);
```

Attribute of signal

```
wire IA; // Output of a combinational circuit is defined as wire
wire IB;
```

There are two types of signal properties:

Attribute	Meaning
wire	output signal of a combination circuit
reg	output signal of a flip-flop (memory element)

Attribute must be declared beforethe signal is being used for the first time (on the top line).
For example, below is a correct example. Notice the declaration position of B.

```
wire B;
wire C;

assign C = B;
assign B = A;
```

For the example below, there will be error:

```
wire C;
```

```
assign C = B;

wire B;

assign B = A;
```

For the above case:

```
assign C = B;
```

B is used, but declared afterwards, so it is an error.
Therefore, a signal has to be declared before it is used.

All the internal signals have to be declared. Internal signals are the signals that are not shown in the port list. Do not declare the input signals that are in the port list.
The way to write the port list in this example has the same meaning as the description below:

```
module MY_XOR2(
  input wire IN_A, // input A
  input wire IN_B, // input B
  output wire O // output O
);
```

The difference is that, a signal attribute ("wire" in this case) is added between the input/output definition and the signal name. In many cases, the attribute for the signal in the port list is wire, so it is fine to omit it.
The signals in the port list are considered that they have already been declared as wires. Therefore, if you declare wires for them later again, it will be a duplicate declaration and causes an error.

[Supplement]
The signals given from the test bench for simulation are defined in reg. You can think of this as assigning a value directly to the memory element. Usually, to give a signal for testing a circuit, the time and value are set only at the change point of a certain signal. Therefore, it is needed to be defined as a memory element in order to specify the value at the specified moment.

Circuit description

From here, up to "endmodule", is where the circuit is described.

We will explain how to write the circuit in later chapter. This document will explain two types of statements: "assign" and "always". About the description for simulation, we will explain the "initial" statement.

There are other expression methods, but they are mainly for convenience, for increasing work efficiency, and for being elegant and good-looking, so we will not cover them. You can learn them by other resources.
First, we will learn the three types of statements, and we also use only these three types in this document.

```
  assign IA = IN_A & ~IN_B;
  assign IB = ~IN_A & IN_B;

  assign O = IA | IB;

endmodule
```

Incorporating lower module

Modules can be made with a hierarchical structure.
For instance, let's see an example where MY_XOR2 is incorporated into the parent module TOP, and it is used as a 3-input XOR.

```
/* An example of incorporating two MY_XOR2 modules*/

module TOP(
  input IN_A, // in
  input IN_B, // in
  input IN_C, // in
  output O, // out
);

wire Z;

MY_XOR2 U1(
  .IN_A (IN_A),
  .IN_B (IN_B),
  .O (Z)
);

MY_XOR2 U2(
  .IN_A (IN_C),
  .IN_B (Z),
  .O (O)
);

endmodule
```

Above "endmodule", MY_XOR2 is incorporated.

[Supplement]

When incorporating the lower module MY_XOR2, we use the Verilog-HDL file MY_XOR2.v describing the circuit of MY_XOR2. There are two ways to load this MY_XOR2.v file: explicitly specifying it using an include statement, or specifying it using a development tool. Currently, the development tools is more often used than the include statement. Here, we will proceed assuming using the development tool.

```
MY_XOR2 U1(
  .IN_A (IN_A),
  .IN_B (IN_B),
  .O (Z)
);
```

MY_XOR2 is the module name (circuit name) and **U1** is the instance name (module number). Next, it is the port list including the TOP module signal and the MY_XOR2 module. The connection relationship in between needs to be defined. In this example, IN_A of MY_XOR2 module is connected to IN_A of TOP module, IN_B of MY_XOR2 module is connected to IN_B of TOP module, and O of MY_XOR2 module is connected to Z of TOP module. Lower module signals are represented by a dot at the left of the signal name. At the right of that, write the name of the signal to be connected enclosed in parentheses, in this case, the signal name of the TOP module. The signal names of lower and upper modules can be different. If the hierarchy is different, it will be recognized as a different signal.

```
MY_XOR2 U2(
  .IN_A (IN_C),
  .IN_B (Z),
  .O (O)
);
```

Similarly, the output from U1 is connected to IN_B, and the output O of U2 is connected to the output O of TOP module.

Instance name

The same instance name cannot be given within the same module (layer). It can be used if they are in different modules (layers).

3. Circuit description

3.1 Combinational circuit

We use assign statement to describe combinational circuits.

AND gates, OR gates, and NOT gates are used to describe combinational circuits. Let's see how to use them:

```
assign output_signal = logic_expression;
```

Assign, output signal name, equal sign, logical expression, and finally ends with a semicolon.
It can be written with multiple lines.
Let's take the example of the 2-input XOR again.

```
assign O = (IN_A & ~IN_B) |
           (~IN_A & IN_B);
```

Please try to write it in a way to understand easily.
We choose some of the frequently used operators in logical expressions to explain below.

Operator	Meaning
&	AND (logical product)
|	OR (logical sum)
~	NOT (inversion)
^	XOR (exclusive logical sum)

Let's see the XOR example again:

```
assign O = (IN_A & ~IN_B) | (~IN_A & IN_B);
```

Operations in parentheses have higher priority. Operators have precedence, so parentheses may not be necessary, but it is recommend to give priority for easier understanding by adding parentheses exaggeratedly.

[Supplement] Precedence of operators
From highest to lowest: NOT, AND, XOR, OR.

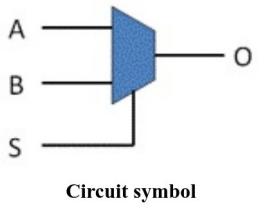
In the above table, this is also "XOR", so you can describe the previous example easily:

```
```

```
assign O = IN_A ^ IN_B;
```

Ternary operator

In this section, we will explain the ternary operator that are useful when writing selectors and multiplexers.
Consider a selector circuit that switches between two inputs A and B using a selection signal S.



Circuit symbol

I/O relation

A	B	S	O
L	X	L	L
H	X	L	H
X	L	H	L
X	H	H	H

X means don't care

If we write this circuit using Verilog-HDL:

```
assign O = (S ? B : A);
```

The parentheses is not needed here.

This selector can be described with this one line. It is used as a data bus selection circuit in combination with the multi-bit representation described next.

[Supplement] Precedence of ternary operator
It is the lowest among the operators.

Multi-bit representation

Signals with identifiers can be also used for multi-bit representations. In digital circuits, multiple signals can be used to represent a value.
For example, addresses and data. 32 bit data has a meaning in 32 lines. Defining such signals one by one is not convenient and also difficult to understand, so it can be expressed with a single identifier as shown below.
Let's see at an example module using multi-bit representation.

```
/* An example of multi-bit representation */
```

```
module DECODER(
  ADDRESS, // in : Input , Address[7:0]
  CE       // out : Output , Chip enable[1:0]
);

// ----- input/ output -----
input [7:0] ADDRESS;
output [1:0] CE;

//-------------------------------------------------------
// Address decoder
//-------------------------------------------------------

wire [1:0] CE; // Output of combination circuit is defined as wire

assign CE[0] = (ADDRESS[7:0]==8'h0);
assign CE[1] = (ADDRESS[7:0]==8'h1);

endmodule
```

Port list

The port list remains the same even if it is multi-bit.
This part only describes the signal names.

Port declaration

```
input [7:0] ADDRESS;
output [1:0] CE;
```

Specify the bit range between the input/output definition and the signal name in [u:l] format.
[u:l] represents the bit range. Separate the MSB on the left side and the LSB on the right side with a colon in between. The values of LSB and MSB are arbitrary.

Supplement
Some people write LSB at left as [l:u], but [u:l] is used mostly.

Here, ADDRESS is declared as an input signal with a bit width of 8, LSB of 0 and MSB of 7, and CE is declared as an output signal of bit width 2, LSB of 0 and MSB of 1.

Internal signal attributes

```
wire [1:0] CE; // Output of combination circuit is defined as wire
```

Specify the bit range [u:l] between the signal attribute and the signal name as in the port declaration.

Circuit description

```
assign CE[0] = (ADDRESS[7:0]==8'h0);
assign CE[1] = (ADDRESS[7:0]==8'h1);
```

Multi-bit signals are often used as numerical values, so not only logical operations, but also conditional extraction (such as equality and inequality signs) may be used as shown above. Also arithmetic operations can be used.

Conditional operators

Operator	Meaning
==	equal
!=	not equal
>	(lhs is) larger
<	(lhs is) smaller
>=	(lhs is) larger or equal
<=	(lhs is) small or equal

When the condition is met, it becomes 1. When the condition is not met, it becomes 0.
`assign CE[0] = (ADDRESS[7:0]==8'h0);`

In this case, **CE[0] becomes 1** only when **ADDRESS[7:0] is 0**.

You can specify any range between the defined LSB and MSB.
For example, from bits 1 to 6 of ADDRESS:

```
assign CE[0] = (ADDRESS[6:1]==6'h0);
```

You can also extract one of the bit from the multi-bit signal:

```
```

```
assign CE[0] = ADDRESS[0];
```

Or

```
assign CE[0] = (ADDRESS[0]==1'b1);
```

In this case, bit 0 is extracted.

Arithmetic operation

Operator	Meaning
+	Addition
-	Subtraction

For instance, we add 1 to the signal A[7:0] into an output signal O[7:0] as sum.

```
wire [7:0] O;
assign O[7:0] = A[7:0] + 8'd1;
```

About carry

In the above example, carries of 9 bits or more are ignored. For example, if you add 8'd1 to A[7:0]=0xFF, the output will be 0x100 if there is carry. But there is no 9th bit, only the digits below 8 bits will be calculated correctly, so the output is O[7:0]=0x0.

Multiplication, division, modulus

Multiplication, division, and modulus operators are also defined but rarely used in circuit descriptions. The scale of these circuits is large, and there are many ways of implementation. To specify a suitable circuit among the many ways, it is necessary to write a description that decomposes the gate level to an imaginable degree, or to use a library specifying the circuit at the gate level.

Simulation

In a simulation test bench, there is no need to convert the description into a circuit. It is easier to use multiplication, division, and modulus operators directly.

Bit combination

Sometimes, we need to define a new multi-bit signal by combining multiple multi-bit signals.

For example, you want to define one signal C[23:0] by combining signal A [7:0] and signal B [15:0]. Connect B[15:0] to C[15:0]. Connect A[7:0] to C[23:16].

We can do it one line by one line as below:

```
wire [23:0] C;

assign C[0] = B[0];
assign C[1] = B[1];
   :    :      :
assign C[15] = B[15];
assign C[16] = A[0];
   :    :      :
assign C[23] = A[7];
```

But this is bothersome. You can also do it in this way:

```
wire [23:0] C;

assign C[15:0] = B[15:0];
assign C[23:16] = A[7:0];
```

Also, it is fine to write it in one sentence:

```
wire [23:0] C;

assign C[23:0] = {A[7:0],B[15:0]};
```

Not only the right side, but also the left side can use this way:

```
wire [15:0] D;
wire [7:0] E;

assign {D[15:0],E[7:0]} = {A[7:0],B[15:0]};
```

3.2. Circuit with memory elements

Before explaning sequential circuit, we will explain circuit descriptions with memory elements. Here, consider D-type flip-flop (DFF) as the memory element. Please see references for other types of memory elements. DFF has more functions than single gate. It has some or all of the following functions:

- reset (synchronous or asynchronous)
- preset (synchronous or asynchronous)
- clock enable

Xilinx encourages to use positive logic signals, so we will start with a description in this way.

D-type Flip Flop (DFF)

This is a basic description of a single DFF without reset or clock enable.
The clock port name is CLK, the input signal is D, and the output signal is Q.

```
reg Q;

always @(posedge CLK) begin
  Q <= D;
end
```

Here is the explanation on this code:

always statement

When the conditional statement written after the keyword "always" inside "@()" is met, the statement between the keywords "begin" and "end" will be executed. In this case, "posedge CLK" is the operation condition. The keyword "posedge" means rising edge. Therefore, this circuit operates every time the rising edge of the clock input CLK occurs. In other words, it operates only when the clock input CLK rises. Otherwise, it will not work, which means that the state is retained.

In summary, for a DFF, the input D is output to Q at the rising edge of CLK. Otherwise Q is retained.

The sign in the middle of the behavior description is "<=" instead of "=". This is not to compare the size. Actually, you can use both "=" and "<=", but the meaning is different.
Remember to use "<=" instead of the equal sign "=" when using it for a DFF.

Difference between "=" and "<=" inside the always statements

When using "=", the result is from evaluating all expressions in the always statement. When using "<=", the results is up to the semicolon. This difference tends to be shown when multiple expressions are written in an always statement.

always statement and DFF

The always statement can also be used to express circuits other than DFF. For example, combinational circuits. However, the always statement is rather complicated to write, and it is difficult to understand the generated circuit. It is safer to design the combinational circuit using the assign statement and to decide DFF output using the always statement.

"begin" and "end"

Keywords o specify the structure of a sentence. Similar to parentheses {} in C language.
In the case of Verilog-HDL, the semicolon is interpreted as one statement, so the above example can be simplified as below.

```
always @(posedge CLK)
  Q <= D;
```

Although we can omit "begin" and "end", if it is important to keep this structure in mind. It is still recommended to use "begin" and "end".
If there are multiple expressions in the always statement, "begin" and "end" are needed.

"posedge" and "negedge"

There are not only "posedge" but also "negedge". Here is an example of using clock falling edge:

```
always @(negedge CLK) begin
  Q <= D;
end
```

Remember to write a DFF in this way until getting used to HDL. Once you get used to it, check the meaning of the sentences again. Sometimes, if you try to strictly connect the meaning and behavior of the always statement, there is difficulty for understanding.

DFF with synchronous reset

The clock port name is CLK, the reset input is RESET, the input signal is D, and the output signal is Q. For the DFF with synchronous reset, an example is as below.

```
reg Q;

always @(posedge CLK)begin
  if(RESET) begin
    Q <= 1'b0;
  end else begin
    Q <= D;
  end
end
```

Here, an if statement is added. When the condition inside () if is met (the condition is true), the following statement is executed. The meaning of this conditional statement (RESET) is the same as ("RESET==1'b1"). When RESET is 1, the following statement below will be executed.

```
    Q <= 1'b0;
```

When this condition inside if is not met, the statement following "else" will be executed:

```
    Q <= D;
```

In summary, at the rising edge of the clock input CLK, the circuit outputs 0 when RESET is 1, and outputs D when RESET is 0 at the rising edge.

DFF with synchronous reset and clock enable

The clock port name is CLK, the clock enable is CE, the reset input is RESET, the input signal is D, and the output signal is Q.
An example is as below.

```
reg Q;

always @(posedge CLK)begin
  if(RESET) begin
    Q <= 1'b0;
  end else begin
    if(CE)begin
      Q <= D;
    end
  end
end
```

From the previous example, ad additional part below "else":

```
    if(CE)begin
      Q <= D;
    end
```

Since this statement is added inside "else", it works when RESET=0. Due to the if statement CE==1, the statement below only works when RESET=0 and CE=1:

```
    Q <= D;
```

There is no description when CE=0, but what will happen when CE=0? If there is no description, the state is retained. In other words, the state is maintained in the stopped state.

Actual description

For all of the above examples, the output is connected to input, but we can also put a combinational circuit at the right side.

For example, a circuit where a DFF is connected to the output of a 2-input XOR:

```
module MY_XOR2_WDFF(
  input CLK  , // in  : Input, System clock
  input RESET, // in  : Input, System reset
  input CE   , // in  : Input, Clock enable
  input IN_A , // in  : Input
  input IN_B , // in  : Input
  output O     // out : Output
);

always @(posedge CLK)begin
  if(RESET) begin
    Q <= 1'b0;
  end else begin
    if(CE)begin
      Q <= (IN_A & ~IN_B) | (~IN_A & IN_B);
    end
  end
end

assign O = Q;

endmodule
```

3.3 Sequential circuit

Here, we will introduce the typical types of circuits using memory elements below.

- Counter
- Shift register

Counter

Counter is a representative of sequential circuits and it is used frequently.
An example of an 8-bit counter with reset and clock enable is shown below.

```
reg  [7:0] Q;

always@ (posedge CLK) begin
  if(RST)begin
    Q[7:0] <= 8'd0;
  end else begin
    if(CE)begin
      Q[7:0] <= Q[7:0] + 8'd1;
    end
  end
end
```

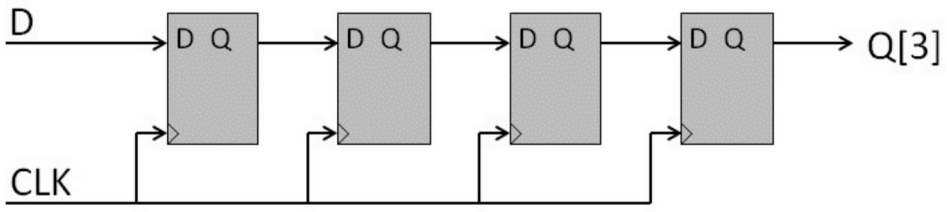
Basically the same description as DFF. The difference is the arithmetic operation "+" and that its own output is returned to the input.
Sometimes, we need to express a carry by intentionally changing the signal name to make the code easier to read. In that case, we can do:

```
reg [6:0] Q;
reg carry;

always@ (posedge CLK) begin
  if(RST)begin
    {carry,Q[6:0]} <= 8'd0;
  end else begin
    if(CE)begin
      {carry,Q[6:0]} <= {carry,Q[6:0]} + 8'd1;
    end
  end
end
```

Shift register

A shift register is a circuit where multiple DFFs are connected in series.



If we directly describe this circuit using HDL:

```
reg  [3:0] Q;

always@ (posedge CLK) begin
  Q[0] <= D;
end

always@ (posedge CLK) begin
  Q[1] <= Q[0];
end

always@ (posedge CLK) begin
  Q[2] <= Q[1];
end

always@ (posedge CLK) begin
  Q[3] <= Q[2];
end
```

This is correct but difficult to read.
You can put it in a single always statement to make it easier to read like below:

```
reg  [3:0] Q;

always@ (posedge CLK) begin
  Q[0] <= D;
  Q[1] <= Q[0];
  Q[2] <= Q[1];
  Q[3] <= Q[2];
end
```

It can be also easily written with multi-bit representation:

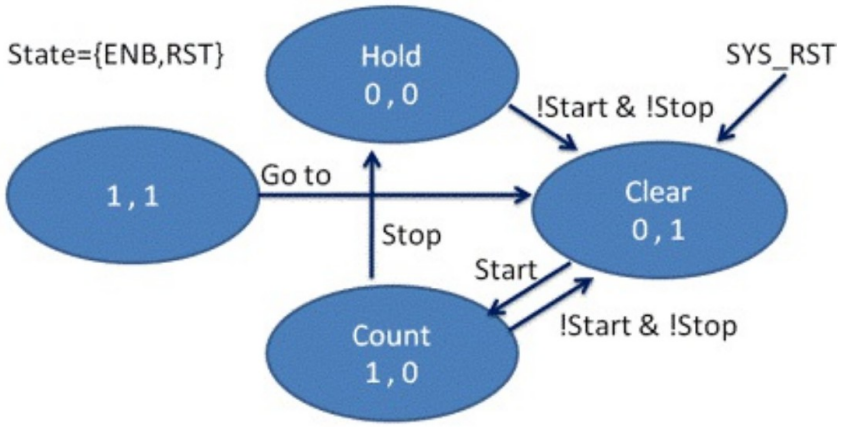
```
reg  [3:0] Q;

always@ (posedge CLK) begin
  Q[3:0] <= {Q[2:0],D};
end
```

This is the typical way to describe a shift register.

State machine

State machine is a useful description to circuits for control purpose.
For instance, consider writing a state machine with the state transition diagram below:



We will explain the operation of this state machine in an easy way.
The number in the circle represents the ate (output state).
When a system reset is input, it becomes "Clear" state with ENB=L, RST=H.
When Start=H, transit to "Count| state with ENB=H, RST=L.
If Stop=H after that, it transit to "Hold" sate with ENB=L, RST=L.
When Start=L and Stop=L, it will return to "Clear" state.
There are several ways to write it in Verilog-HDL., Here we will use the case statement.

```
wire RST;
wire ENB;
reg [1:0] State;

always@ (posedge CLK) begin
  if(SYS_RST)begin
    State[1:0] <= 2'b01;
  end else begin
    case(State[1:0])
      2'b01:begin Clear
        if(Start)begin
          State[1:0] <= 2'b10;
        end else begin
          State[1:0] <= 2'b01;
        end
      end

      2'b10:begin // Count
        if(!Start & !Stop)begin
          State[1:0] <= 2'b01;
        end else if(Stop)begin
```

```
          State[1:0] <= 2'b00;
        end else begin
          State[1:0] <= 2'b10;
        end
      end

      2'b00:begin // Hold
        if(!Start & !Stop)begin
          State[1:0] <= 2'b01;
        end else begin
          State[1:0] <= 2'b00;
        end
      end

      default:begin
        State[1:0] <= 2'b01;
      end
    endcase
  end
end

assign {ENB,RST} = State[1:0];
```

4. Simulation description

To validate that the designed circuit's operation, a simulator on a computer is needed before implementing the circuit.
Simulation can be simply categorized into logical simulation and delay simulation. Logic simulation calculates logic behavior without considering the delay of circuit operation (assuming infinitesimal delay). This is only to validate the circuit operation.
Delay simulation is a method with circuit delay information.

Here, we will explain only the logical simulation.

Delay simulation: When doing FPGA design, we rarely do delay simulation. It is possible to enhance your understanding on the operation by performing simulation, so please try to do it.
Sometimes people worry the difference of results between simulation and real device, but it's natural to be working in simulation. Since the circuit is designed to work in the simulation environment created by the designer, it will definitely work. When a problem occurs, a so-called bug happens because the situation (signal behavior) is unexpected. Therefore, it is important to familiarize yourself with the input signals of the real device and to perform tests under as many different simulation conditions as possible.

Things needed for simulation

In addition to the designed Verolog-HDL file (DUT), we need the following items to perform simulation.

- Simulator
- Test bench (Verilog-HDL file)

Simulator
There are various available simulators. If you search for Verilog on Wikipedia, some will be listed. Here, we will proceed with the discussion with Veritak.

Test bench
A test bench describes methods such as generating the signals (to be given to the DUT) necessary to validate the designed circuit (DUT) and displaying results for efficient validation. Since the test bench does not need to be converted into a circuit, we should put higher priority to write it efficiently and make it easily understood. Therefore, some of the description methods which cannot be used in a circuit can be used in test bench.

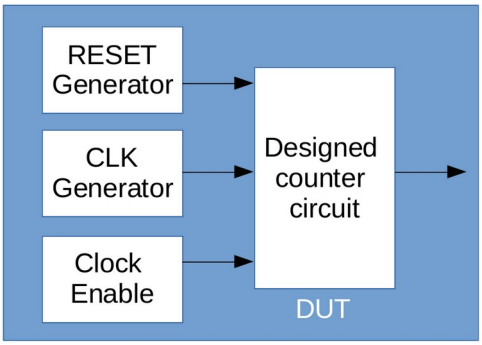
Test bench

Let's assume that we have an FPGA circuit in HDL and would like to simulate it to validate it.
The test bench is also a Verilog-HDL module. It is written in text just like a module.

The designed circuit is for FPGA and is part of the PCB board. Transmitter, reset switch, LED, and external connector are mounted on the board. To operate and verify the designed circuit, it is necessary

to give signals to the circuit to imitate the operation of those external components. In the case of a synchronous circuit, a clock is needed, and a reset signal must be given to determine the initial state. A test bench is used to generate signals to simulate actual operation.

Here is an example of a test bench to verify a counter design:



Test bench

The designed counter has three inputs: reset, clock and clock enable. To test the counter circuit, we need to provide these three inputs.

There is no specific way to write a test bench. For example, since external memory is used, it is not necessary to write a circuit (model) to imitate the external memory on the test bench. Of course, if you can get a perfectly working model, it is better to connect it and verify it. But in reality, you have to make the model yourself. It will increase the development and verification time. Validating the model behavior is particularly important. If the model is wrong, the verification results will also be wrong. The circuits designed by us are usually not that complicated compared to those designed by companies. Therefore, instead of just assembling a model properly, the designer should think carefully about what to verify and how to verify it to consider it as working, and then devise and give inputs to the circuit to be tested. Considering the required development period, it is almost impossible to completely verify everything by simulation.

Considering the development period, it is basically impossible to create a perfect environment imitating actual signals completely, and it is also difficult to perform long-term simulation. One important thing of doing simulation is a test bench that can achieve the best verification effect with the minimal effort. If you have a thorough understanding on the circuit you designed, you should know what kind of signals will make it behave unreliably, and what kind of behavior will happen when an unexpected signal is given.
Be sure to write a test bench that allows designers to efficiently verify your concerns on the circuit.

Description specific for simulation

Test bench is a Verilog-HDL module. But different from other modules, no need to convert it to a circuit. Therefore, you can write without considering real circuit. You can use all the valid descriptions in syntax.

Next, we will provide a practical example.

4.1 Counter simulation

Here, we will explain using an example of a 8-bit counter.

Circuit to test

The circuit to be verified is the coutner below:

```
/* 8-bit counter */

module COUNTER(
  CLK, // in : System clock
  RST, // in : System reset
  CE, // in : Clock enable
  Q // out : Count[7:0]
);

//-------- Input/Output -------------
input CLK ;
input RST ;
input CE ;
output [7:0] Q ;

//-------- 8-bit counter -------------
reg  [7:0] Q ;

always@ (posedge CLK) begin
  if(RST)begin
    Q[7:0] <= 8'd0;
  end else begin
    if(CE)begin
      Q[7:0] <= Q[7:0] + 8'd1;
    end
  end
end

endmodule
```

Below is the post list of this module:

```
input CLK ;
input RST ;
input CE ;
output [7:0] Q ;
```

Without giving the input signals to CLK, RST, and CE, the circuit will not work.
Therefore, we need to generate them in the test bench.

Test bench

This is written with the minimal necessary functions:

```
/*****************************************************************
*
* Test bench for counter
*
*****************************************************************/

`timescale 1ps/1ps

`include "COUNTER.V"

module COUNTER_TB;

reg  OSC50M ;
reg CLKENB ;
reg RST ;
wire [7:0] Q ;

//---------------------------------------------------------------
// DUT
//---------------------------------------------------------------

COUNTER DUT(
// System
  .CLK (OSC50M ), // in : System clock
  .RST (RST ), // in : System reset
  .CE (CLKENB ), // in : Clock enable
  .Q (Q[7:0] ) // out : Count[7:0]
);

//---------------------------------------------------------------
// Clock generator
//---------------------------------------------------------------

parameter OSC50M_PERIOD = 20000; // ps

initial begin
  OSC50M = 1'b0;
end
```

```
always #(OSC50M_PERIOD/2) begin
  OSC50M <= ~OSC50M;
end

reg [1:0] clkEnbCntr ;

initial begin
  CLKENB = 1'b0;
  clkEnbCntr = 2'b0;
end

always@ (posedge OSC50M) begin
  clkEnbCntr[1:0] <= clkEnbCntr[1:0] + 2'd1;
  CLKENB <= (clkEnbCntr[1:0]==2'd1);
end

//---------------------------------------------------------------------
// Reset generator
//---------------------------------------------------------------------

initial begin
  RST = 1;
  repeat(20) @(negedge OSC50M);
  RST = 0;
end

//---------------------------------------------------------------------
// Test vector
//---------------------------------------------------------------------

initial begin
  @(negedge OSC50M);
  while(RST) @(negedge OSC50M);
  repeat(100) @(negedge OSC50M);
  $finish;
end

//---------------------------------------------------------------------

endmodule
```

Some of the descriptions appear for the first time. We will explain them in order.

Environment setting

For a test bench, it starts from "module" to "endmodule". The sentences above the keyword "module" are environment settings. In this example, we specify the time unit for simulation and the designed circuit:

```
`timescale 1ps/1ps

`include "COUNTER.V"
```

"`timescale" is the keyword. Simulations are performed in a discrete time unit. Set the time unit based on the designed circuit. "1ps/1ps" following the keyword represents unit time and rounding precision. Usually, we set them as the same value.

Xilinx FPGA

Many circuit libraries provided by Xilinx assume a timescale of 1ns/1ps. It is also safe to use 1ns/1ps setting is safe when using Xilinx FPGA without using a library.

```
`include "COUNTER.V"
```

This is to specify the designed circuit. Some simulators may not work the "`include" statement is not used in the simulator's project file. Please set it according to your environment. Here, we will proceed using Veritak.

Declaration of module and internal signal

```
module COUNTER_TB;

reg OSC50M ;
reg CLKENB ;
reg RST ;
wire [7:0] Q ;
```

Similar to the other circuits explained ealier, test bench is also a module, so the description of its operation is put from "module" to "endmodule".
Different from the designed circuit, there is no port list. Because this module is the closed top layer and it describes the environment where the developed circuit will operate.

Next is the description of the internal signals. The input signals to the counter circuit to be tested are clock, clock enable, and reset. Since the test bench generates those signals, they are declared as a memory element (reg). The reason is that, the time and value of those signals need to be defined. For example, the clock is low at time 0, high at 10ns, low at 20ns, etc. To specify the time and value of the change point, and keep the value other than the change point, we need to use them as memory element,

although the way to set values in test bench is not a logic circuit. You can think of it as a test bench memory element that can be directly set.
Finally, there is a wire declaration for the counter's output signal.

Incorporating the designed circuit

The design circuit to be tested is called a Device Under Test (DUT).
The way to incorporate it is the same as a normal circuit.

```
//---------------------------------------------------------------------
// DUT
//---------------------------------------------------------------------

COUNTER DUT(
// System
  .CLK (OSC50M ), // in : System clock
  .RST (RST ), // in : System reset
  .CE (CLKENB ), // in : Clock enable
  .Q (Q[7:0] ) // out : Count[7:0]
);
```

Clock generation circuit

There is a commonly used description for clock generation circuits. You can find it at the bottom part:

```
//---------------------------------------------------------------------
// Clock generator
//---------------------------------------------------------------------

parameter OSC50M_PERIOD = 20000; // ps

initial begin
  OSC50M = 1'b0;
end


always #(OSC50M_PERIOD/2) begin
  OSC50M <= ~OSC50M;
end
```

Setup for clock period

```
```

```
parameter OSC50M_PERIOD = 20000; // ps
```

The keyword "parameter" is constant declaration. OSC50M_PERIOD is defined as 20000. The simulation environment has been set to 1 ps as a time unit, so it corresponds to 20 ns.
You can use the number directly in the always statement below, but we use constants to make the code easier to read. If you do this, it will be an oscillator with a period of 50M.

Initialization of the clock signal

```
initial begin
  OSC50M = 1'b0;
end
```

It is initialized with the initial statement, which is an instruction executed only once when the simulator starts, and is often used on test benches. Here, there is only one line: "OSC50M = 1'b0;", so it will be set to 0 in the beginning of the simulation. Since OSC50M is a memory element output, it will remain 0 if nothing is done after initialization.

Notice for synthesizing circuits
Do not use initial statements in actual circuit descriptions. If you really want to use it, please read the synthesis tool's manual carefully before using it.

Clock generation

```
always #(OSC50M_PERIOD/2) begin
  OSC50M <= ~OSC50M;
end
```

The "always" here is a little different from the ones we have used before. The character that used to be "@" has changed to "#".
"@" executes when the condition is met, but "#" waits only for the value written after "#", in this case OSC50M_PERIOD/2. It will always wait for the value of "#", so it will be executed in every 10ns. Next, we set the output to the inverted value of the OSC50M output.

For the result, the initial value is 0, and then the value is inverted in every 10ns, so a clock has been generated.

Clock enable generation circuit

For the clock enable generation circuit, the clock enable will be set as H in every 4 clock-cycles of OSC50M.

```
reg [1:0] clkEnbCntr ;
```

```
initial begin
  CLKENB = 1'b0;
  clkEnbCntr = 2'b0;
end

always@ (posedge OSC50M) begin
  clkEnbCntr[1:0] <= clkEnbCntr[1:0] + 2'd1;
  CLKENB <= (clkEnbCntr[1:0]==2'd1);
end
```

It's the same as the case when generating clock.

First, we declare a dividing counter signal to generate a clock enable. After that, initialization is done with the initial statement, and the always statement is used in the same way as a normal circuit. As from the clock generation circuit, for the description of an actual circuit to be design, do not use the way of initializing with the initial statement and describing the circuit operation with the always statement.

Reset generation circuit

It is also possible to write the reset generation in the test vector, but here we try to do it separately from the test vector.

```
//---------------------------------------------------------------------
// Reset generator
//---------------------------------------------------------------------

initial begin
  RST = 1;
  repeat(20) @(negedge OSC50M);
  RST = 0;
end
```

Since it is an initial statement, the setences in the statement are executed only once from top to bottom after the simulator starts.
Initially, RST = 1, so set the RST signal to high.

```
  repeat(20) @(negedge OSC50M);
```

The keyword "repeat(20)" is a repeat command. It repeats the next command @(negedge OSC50M) 20 times by the number in parentheses.

@(negedge OSC50M) is a condition that operates only at the falling edge of OSC50M, so this statement repeats until it finds OSC50M falling 20 times, so it is an instruction of waiting for 20 OSC50M clocks.

Next, RST = 0, so the RST signal is set to L.

The above description simulates a circuit which resets the OSC50M for 20 clocks after the simulator starts.

Test vector

```
//---------------------------------------------------------------------
// Test vector
//---------------------------------------------------------------------

initial begin
  @(negedge OSC50M);
  while(RST) @(negedge OSC50M);
  repeat(100) @(negedge OSC50M);
  $finish;
end
```

Here we will explain the "while" and "$finish".
For "while". it executes statement @(negedge OSC50M) while the condition inside the parentheses is met. When RST=H, it keeps waiting for the falling edge. After RST=L, it executes the next line at the falling edge immediately.

"$finish" is the command to end simulation. In the case of Veritak, without it, it will continue running and will never stop.

Simulation result

The simulation result from Veritak is shown below, where you can see that the counter is working correctly.